**Unix Scripting**

Shell: shell is an interface to communicate with Unix/Linux system. It gather input from user and execute program based on that input.

Shell is an environment in which we can run our commands, programs, and shell scripts.

**Shell Types**

In Unix, there are two major types of shells −

* **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
* **C shell** − If you are using a C-type shell, the % character is the default prompt.

**Shell Types**

In Unix, there are two major types of shells −

* **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
* **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh)

The different C-type shells follow −

* C shell (csh)
* TENEX/TOPS C shell (tcsh)

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

## Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

Shell scripts and functions are both interpreted. This means they are not compiled.

Example script:

Bash Shell script starts with #!(shebang)

Always start with #! /bin/bash

***#!/bin/bash***

***#author: chakra Neopaney***

***#script follows here***

***Pwd***

***Ls***

Save the above script as test.sh and make it executables.

***$chmod +x test.sh***

The shell script is now ready to be executed.

***$./test.sh***

Upon execution you will receive the following results.

*/home/amrood 🡪pwd*

*index.htm unix-basic\_utilities.htm unix-directories.htm ----🡪ls*

*test.sh unix-communication.htm unix-environment.htm---🡪ls*

**Note** − To execute a program available in the current directory, use **./program\_name**

**Using variables**

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as **!**, **\***, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows −

variable\_name=variable\_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"

VAR2=100

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

[Live Demo](http://tpcg.io/AP7zgT)

#!/bin/sh

NAME="Zara Ali"

echo $NAME

The above script will produce the following value −

Zara Ali

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

[Live Demo](http://tpcg.io/tawT1C)

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types

When a shell is running, three main types of variables are present −

* **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
* **Environment Variables** − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
* **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

**SPECIAL VARIABLES**

For example, the **$** character represents the process ID number, or PID, of the current shell −

$echo $$

The above command writes the PID of the current shell −

29949

The following table shows a number of special variables that you can use in your shell scripts −

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  The process number of the last background command. |

## Command-Line Arguments

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

## Special Parameters $\* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$\*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $\* or $@ special parameters −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

Here is a sample run for the above script −

$./test.sh Zara Ali 10 Years Old

Zara

Ali

10

Years

Old

**Note** − Here **do...done** is a kind of loop that will be covered in a subsequent tutorial.

## Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

$./test.sh Zara Ali

File Name : ./test.sh

First Parameter : Zara

Second Parameter : Ali

Quoted Values: Zara Ali

Quoted Values: Zara Ali

Total Number of Parameters : 2

$echo $?

0

$

**USING Shell ARRAY**